HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Unterstützung von Joi als Programmierstil mit Annotationen und Checkstyle

Jonas Middendorf

Konstanz, 18.06.2010

BACHELORARBEIT

Bachelorarbeit

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

an der

Hochschule Konstanz
Technik, Wirtschaft und Gestaltung

Fakultät Informatik

Studiengang Wirtschaftsinformatik

Thema: **Unterstützung von Joi als Programmierstil mit Annotationen und Checkstyle**

Bachelorkandidat: Jonas Middendorf
Höhrenbergstr. 2
78476 Allensbach

1. Prüfer: Prof. Dr. Heiko von Drachenfels
2. Prüfer: Prof. Dr. Oliver Haase

Ausgabedatum: 18.03.2010
Abgabedatum: 18.06.2010

Zusammenfassung (Abstract)

Thema:	Unterstützung von Joi als Programmierstil mit Annotationen und Checkstyle
Bachelorkandidat:	Jonas Middendorf
Firma:	HTWG-Konstanz
Betreuer:	Prof. Dr. Heiko von Drachenfels Prof. Dr. Oliver Haase
Abgabetermin:	18. Juni 2010
Schlagworte:	Joi, Java, Checkstyle, Programmierstil, Komponenten, Annotationen, Namenskonventionen, Komposition, Singleton, Factory

Die vorliegende Arbeit untersucht, wie Tools für die Überprüfung von Quell-Code auf Konformität mit Programmierstilen, die komplexere Entwurfsmuster beinhalten, als Erweiterung des statischen Code-Analyse-Tools Checkstyle realisiert werden können. Als Beispiel wird der Joi Programmierstil verwendet, der für diese Arbeit aus der Java Spracherweiterung Joi (Java Objects by Interfaces), die die Einhaltung einiger Entwurfsmuster, die auf die Reduzierung von Code-Abhängigkeiten zielen, unterstützt, abgeleitet wurde.

Danksagung

Mein aufrichtiger Dank gilt meinen Betreuern Herrn Prof. Dr. Heiko von Drachenfels und Herrn Prof. Dr. Oliver Hasse für die engagierte und anregende Betreuung bei der Erstellung dieser Arbeit.

Inhaltsverzeichnis

1 Einleitung

Joi (Java Objects by Interfaces) ist eine Java Spracherweiterung, die die Einhaltung einiger Entwurfsmuster und Programmierrichtlinien unterstützt, die auf die Reduzierung von Code-Abhängigkeiten zielen. (vgl. Drachenfels[3], Drachenfels et. al. [4])

Eine Spracherweiterung wie Joi erfordert die Unterstützung durch einen Compiler (oder Interpreter). Die Bereitstellung eines Compilers, der in der Praxis angenommen wird, stellt eine beachtliche Hürde für die Verbreitung einer Spracherweiterung wie Joi dar.

Die Ziele, die mit Joi verfolgt werden, können statt mit einer Spracherweiterung auch mit einem Programmierstil erreicht werden - Programmierstil in dem weiteren Sinne verstanden, der die Beachtung ausgewählter Entwurfsmuster (Design Pattern) einschließt.

Ein Programmierstil benötigt keinen speziellen Compiler. Wird Joi also als Programmierstil formuliert, fällt also die ‚Compiler-Hürde' weg.

Einem Programmierstil kann man grundsätzlich ohne spezielle Unterstützung durch Entwicklungs-Tools folgen. Für den Erfolg eines Programmierstils in der Praxis ist geeignete Toolunterstützung jedoch entscheidend. Nur wenn die Einhaltung der Regeln des Stils automatisch überprüft werden kann, kann auch ein Erfolg im Hinblick auf ein Ziel, wie die Reduzierung von Codeabhängigkeiten, abgesichert werden.

Checkstyle ist ein verbreitetes Tool für statische Codeanalyse zur Prüfung der Einhaltung von Programmierrichtlinien. Checkstyle wird ursprünglich für die Prüfung einfacherer Aspekte von Programmierstilen wie Namenskonventionen und Formatierungsrichtlinien eingesetzt. (vgl. Burn [2])

Annotationen können zur Strukturierung von Quellcode verwendet werden. Sie können Programmierelemente wie Klassen, Methoden, Parameter und Attribute kennzeichnen und Metainformationen bereitstellen, die zum Beispiel eine Semantische Bedeutung in einem Programmierstil haben. Tools, wie Compiler oder Checkstyle können diese Metainformationen auslesen und auswerten.

In dieser Arbeit wird untersucht, wie Joi als Programmierstil mit Annotationen und Checkstyle unterstützt werden kann. Der Weg, Joi als Programmierstil mit Checkstyle als Standardtool zu propagieren, könnte eine Alternative zu dem Weg einer Java Spracherweiterung mit speziellem Compiler sein.

Im Folgenden wird zunächst die Formulierung von Joi als Programmierstil behandelt. Neben einen Regelsatz, der die Aspekte von Joi möglichst vollständig als Programmierstil umsetzt (Joi-Stil) wird eine im Hinblick auf Praxistauglichkeit vereinfachte Variante JoiLight gestellt.

Dann wird die Verwendung von Annotationen und Namenkonventionen zur Markierung von Strukturelemente erläutert und anschließend die Realisierung der Prüfung des Joi-Stils mit Checkstyle erläutert.

Im Folgenden wird Joi und die Motivation für Joi als bekannt vorausgesetzt. Zur Einführung sei auf Drachenfels[3] und Drachenfels et al. [4] verwiesen.

2 Joi als Programmierstil

Für die vorliegende Arbeit wurde eine Formulierung von Joi als Programmierstil erarbeitet, die die Überprüfung mit Checkstyle ermöglicht

Der Joi-Stil beschreibt die Programmierung von Joi-Komponenten. Die Grundaspekte des Joi Konzepts können wie folgt zusammengefasst werden:

- Objekte sollen ausschließlich über die Schnittstellentypen referenziert werden, die von den Klassen, die die Objekte beschreiben, implementiert werden.
- Zu jeder nicht privaten Methode soll es eine Schnittstellendefinition geben. Das bedeutet, dass in einer Joi-Komponente keine static Methoden, mit Ausnahme von private static Methoden, existieren dürfen.
- Die Aufgaben der Erzeugung und Beschreibung, bzw. Implementierung von Objekten sollen streng getrennt werden.

Zur Umsetzung dieser Aspekte wird eine Joi-Komponente als eine Outer-Most-, nicht instanziierbare Klasse, die mit der Annotation @JoiComponent gekennzeichnet ist, realisiert und die folgende Elemente enthält:

- eine Klasse Implementation, die die Objekte beschreibt, als private static member;
- ein Enum Factory, die die Methoden zur Erzeugung der von der Implementation beschrieben Objekte beschreibt, als private static member;
- eine Instanz (FACTORY) der Factory, als public static Member;

Joi-Komponenten können zusätzlich um eine spezielle Form der Komposition erweitert werden, für die die „Ergänzenden Regeln für Joi-Komponenten mit Forward Schnittstelle und Forwarding Klasse“ gelten.

2.1 Joi-Grundregeln

In diesem Abschnitt werden die Grundregeln für die Bestandteile der Joi-Komponenten (eine Klasse die mit @JoiComponent gekennzeichnet ist) aufgelistet.

Zur Illustration werden die Regeln ergänzend in einem Code-Beispiel dargestellt. Die Joi-Komponente MyComponent der Code-Beispiele, unterstützt die Service Schnittstelle(Interface) MyService (bedeutet die Implementation Klasse garantiert die Implementierung der Schnittstelle MyService) und die Factory der Joi-Komponente implementiert die Factory Schnittstelle MyProvider:

```
//Service Interface
public interface MyService {
    MyService doSomething ();
}
```

```
//Factory Interface.
public interface MyProvider {
    MyService getInstance();
}
```

2.1.1 Basisregeln für Joi-Komponenten

- Eine Joi-Komponente muss eine Outer-Most-Type-Klasse sein.
- Eine Joi-Komponente muss final sein.
- Eine Joi-Komponente darf von keiner Klasse erben.
- Eine Joi-Komponente darf keine Schnittstelle implementieren.
- Der Default-Konstruktor einer Joi-Komponente muss als private deklariert sein.
    - Ein guter Stil wäre es außerdem, wenn der Konstruktor als einzige Anweisung einen AssertionError() wirft. Dies wird nicht kontrolliert, sondern empfohlen. (vgl. Bloch [1] S. 19)
- In einer Joi-Komponente muss, als Member der Joi-Komponente, eine Klasse mit dem Namen Implementation definiert werden.
- In einer Joi-Komponente muss, als Member der Joi-Komponente, ein Enum mit dem Namen Factory definiert werden.
- In einer Joi-Komponente muss, als Member der Joi-Komponenten, eine Variable mit dem Namen FACTORY definiert werden.
- Eine Joi-Komponente darf keine weiteren Mitglieder haben. (Erweiterung Forward)

Folgendes Code-Beispiel stellt die Joi-Komponente MyComponent da. Aspekte die von den „Basis Regeln für Joi-Komponenten“ beschrieben werden, sind hervorgehoben:

```
//Joi-Component
@JoiComponent
public final class MyComponent extends implements{

        private MyComponent() {
            throw new AssertionError();
    }

        private enum Factory implements MyProvider {…

        public static final Factory FACTORY = Factory.INSTANCE;

        private static final class Implementation implements MyService {…

        …//other members are prohibited
}
```

2.1.1.1 Regeln für die Klasse Implementation einer Joi Komponente

- Die Implementation muss private sein.
- Die Implementation muss static sein.
- Die Implementation muss final sein.
- Die Implementation darf von keiner Klasse erben.
    - Dies gilt nicht wenn in der Joi-Komponente eine Klasse Forwarding definiert ist, in diesem Fall muss die Implementation die Klasse Forwarding beerben.
- Jede public Methode der Klasse Implementation muss mit @Override annotiert sein.

Folgendes Code-Beispiel stellt die Member-Klasse Implementation der Joi-Komponente MyComponent da. Aspekte die von den „Regeln für die Klasse Implementation einer Joi-Komponente“ beschrieben werden, sind hervorgehoben:

```
…//outer class MyComponent
private static final class Implementation extends Forwarding implements MyService {

    @Override
    public void doSomething() {...

     ... // other service-specific members go here

}
…//outer class MyComponent
```

2.1.1.2 Regeln für das Enum Factory einer Joi- Komponente

Da jede Joi-Komponente eine eigene Factory hat und von dieser genau eine Instanz benötigt, bietet es sich an, die Factory als Singleton zu implementieren.

Für den Joi-Stil verwenden wir ein Enum zur Implementierung eines Singletons. Die zentrale Anforderung an ein Singleton, dass es nicht mehr als eine Instanz geben darf, kann mit Enums sehr einfach umgesetzt werden, indem für diese nur eine Konstante als Member definiert wird(vgl. Bloch [1] S.17f).

Diese Implementierung für ein Singleton hat gegenüber der Alternative einer Implementierung als private static Member-Klasse den Vorteil, dass weniger Regeln definiert und geprüft werden müssen. Die Regeln, dass die Factory static und final sein muss und dass sie nicht erben darf, können entfallen, da in Java Member-Enums diese Anforderung immer erfüllen. (Gosling et al. [6] Kapitel 8.9 Enums)

- Die Factory muss private sein.
- Das Factory muss genau eine Konstante mit dem Name INSTANCE enthalten.
- Jede public Methode der Factory muss mit @Override annotiert sein.
- Jede public Methode der Factory muss den Rückgabetyp Implementation haben.
- Die Return-Anweisung jeder public Methode der Factory muss die Gestalt:
  return new Implementation(<parameter>);
  haben.
  - Alle Parameter jeder public Methode müssen in derselben Reihenfolge wie in der Methodendeklaration an den Implementation Konstruktor in der return Anweisung, wiederum als Parameter, übergeben werden.
  - Die Parameter der Methodendeklarationen dürfen nicht mehr sein, als an den jeweiligen Implementation Konstruktor in der return Anweisung, übergeben werden.

Folgendes Code-Beispiel stellt das Member-Enum Factory der Joi-Komponente MyComponent da. Aspekte die von den „Regeln für das Enum Factory einer Joi-Komponente“ beschrieben werden, sind hervorgehoben:

```
…//outer class MyComponent
private enum Factory implements MyProvider {
    INSTANCE;

    @Override
    public Implementation getInstance() {
        return new Implementation();
    }
}
…//outer class MyComponent
```

2.1.1.3 Regeln für die Variable FACTORY einer Joi-Komponente

- Die FACTORY muss public sein.
- Die FACTORY muss static sein.
- Die FACTORY muss final sein.
- Die FACTORY muss vom Typ Factory sein
- Die FACTORY muss mit der Factory Konstanten Factory.INSTANCE initialisiert werden.

Folgendes Code-Beispiel stellt die Member-Variable FACTORY der Joi-Komponente MyComponent da. Aspekte die von den „Regeln für die Variable FACTORY einer Joi-Komponente“ beschrieben werden, sind hervorgehoben:

```
…//outer class MyComponent
public static final Factory FACTORY = Factory.INSTANCE;
…//outer class MyComponent
```

2.2 Erweiterung des Joi-Stils für die Komposition

Der Joi Stil schließt die Vererbung für Joi-Komponenten aus. Für den Fall, dass eine Joi-Komponente Eigenschaften anderer Objekte annehmen soll, wie es die Vererbung ermöglicht, bietet der Joi-Stil als Alternative zur Vererbung eine spezielle Form der Komposition (Joi-Komposition). (vgl. Bloch [1] S. 81ff, Drachenfels et al. [4] S. 1, Gamma et al. [5] S. 395ff)

Die Komposition kann bei Einhaltung dieser Regeln genauso komfortabel genutzt werden wie die klassische Vererbung. Mit der Komposition kann man jedoch einige Probleme der Vererbung vermeiden, wie z.B.“ Das Problem der instabilen Basis-Klasse“.

Der Joi-Stil sieht für die Komposition eine Schnittstelle Forward und eine Klasse Forwarding vor. Wenn die Joi-Komposition verwendet wird, erbt die Klasse Implementation von der Klasse Forwarding.

2.2.1 Ergänzende Regeln für Joi-Komponenten mit Joi-Komposition

- In einer Joi-Komponente kann eine Schnittstelle mit dem Namen Forward definiert werden.
- In einer Joi-Komponente muss genau dann eine geschachtelte Klasse Forwarding definiert werden, wenn auch eine Schnittstelle Forward in der Joi-Komponente definiert wurde
- Die geschachtelte Klasse Implementation muss von der Klasse Forwarding erben.

Folgendes Code-Beispiel stellt die Joi-Komponente MyComponent aus den vorhergehenden Beispielen da, welche um die Joi-Komposition erweitert wurde. Aspekte die von den „Ergänzenden Regeln für Joi-Komponente mit Joi-Komposition“ beschrieben werden, sind hervorgehoben:

```
@JoiComponent
public final class MyComponent ~~extends implements~~{
    private MyComponent() {
        throw new AssertionError();
    }
    private enum Factory implements MyProvider {…

    public static final Factory FACTORY = Factory.INSTANCE;

    private static final class Implementation extends Forwarding implements MyService {…

    public interface Forward {…

    private static class Forwarding implements MyOtherService, Forward {…
}
```

2.2.2 Regeln für die Member-Schnittstelle Forward einer Joi-Komponente mit Joi-Komposition

- Die Schnittstelle Forward muss public sein
- Der Rückgabetyp jeder Methodensignaturen innerhalb der Schnittstelle Forward muss der Typ Forward sein.
- Der Name jeder Methodensignatur innerhalb der Schnittstelle Forward muss forwardee sein.
- Die Methodensignaturen innerhalb der Schnittstelle Forward haben genau einen Parameter. (Dieser muss ein Interface-Typ sein. Wird durch andere Regeln implizit geprüft)

Folgendes Code-Beispiel stellt die Member-Schnittstelle Forward der erweiterten Joi-Komponente MyComponent da. Aspekte die von den „Regeln für die Member-Schnittstelle Forward einer Joi-Komponente mit Joi-Komposition“ beschrieben werden, sind hervorgehoben.

```
…//outer class MyComponent
public interface Forward {
    Forward forwardee(MyOtherService aService);
    …//other forwardee Methods with other parameters
}
…//outer class MyComponent
```

2.2.3 Regeln Für die Member-Klasse Forwarding einer Joi-Komponente mit Joi-Komposition

- Die Klasse Forwarding muss private sein.
- Die Klasse Forwarding muss static sein.
- Die Klasse Forwarding darf nicht final sein.
- Die Klasse Forwarding darf von keiner Klasse erben.
- Alle public Methoden der Klasse Forwarding müssen mit @Override annotiert sein.
- Die Klasse Forwarding darf keinen Konstruktor definieren. Der Default Konstruktor darf nicht überschrieben bzw. überladen werden.
- Die Klasse Forwarding garantiert die Schnittstelle Forward zu implementieren.
- Die Klasse Forwarding muss garantieren die Methoden aller Typen zu implementieren, die als Parameter in einer Methodensignatur der Schnittstelle Forward vorkommenden.
- Die Klasse Forwarding muss für jeden als Parameter in einer Methodensignatur der Schnittstelle Forward vorkommenden Typ ein Attribut desselben Typs haben.
- In der Klasse Forwarding sind Attribute immer über den this Zeiger zu referenzieren.
- Methoden der Klasse Forwarding, die Methodensignaturen der Schnittstelle Forward überschreiben, müssen die Gestalt:

```
@Override
public final Forward forwardee(final <a_parameter>) {
    if (this.<an_attribut> != null) {
        throw new IllegalStateException();
    }
    this.<an_attribut> = <a_parameter>;
    return this;
}
```

haben.

Alle anderen Methoden müssen die Gestalt:

```
@Override
public void <an_identifier>(<parameters>) {
    this.<an_attribut>.<an_identifier>(>(<parameters>);
}
```

Oder:

```
@Override
public <type> .<an_identifier> (<parameters>) {
    (return) this.<an_attribut>.<an_identifier>(<parameters>)
}
```

haben.

Folgendes Code-Beispiel stellt die Member-Klasse Forwarding der erweiterten Joi-Komponente MyComponent da. Aspekte die von den „Regeln für die member-Klasse Forwarding einer Joi-Komponente mit Joi-Komposition“ beschrieben werden, sind hervorgehoben:

```
…//outer class MyComponent
private static class Forwarding extends implements MyOtherService, Forward {
    MyOtherService mService;

    @Override
    public final Forward forwardee(final MyOtherService aService) {
        if (this.mService != null) {
            throw new IllegalStateException();
        }
        this.mService = aService;
        return this;
    }

    @Override
    public void doSomethingElse(String aString) {
        this.mService.doSomethingElse(aString);
    }

    ... // other service-specific methods go here */
}
…//outer class MyComponent
```

2.3 Joi-Singleton-Komponente

Der Joi Stil unterstützt explizit das Singleton- Entwurfsmuster. Dazu wird eine Variante der Joi-Komponente, die Joi-Singleton-Komponente eingeführt.

Für Joi-Singleton Komponenten sind zwei Ausprägungen mit jeweils spezifischen Regeln vorgesehen. Joi-Singleton-Komponenten werden wie alle Joi-Komponente mit der Annotation @JoiComponent gekennzeichnet. Sie unterscheiden sich von normalen Joi-Komponenten dadurch, dass sie statt einer Member-Klasse Implementation ein Member-Enum Implementation enthalten. Wie schon bei der Factory (siehe oben) wird wieder ein Enum zur Implementierung eines Singletons gewählt.
(vgl. Bloch [1] S. 17f)

Der Joi-Stil sieht zwei Varianten von Joi-Singleton-Komponenten vor. Die eine Variante hat keine Factory und anstelle der Variablen FACTORY vom Typ Factory, eine Variable mit dem Typ SINGLETON vom Typ Implementation als Member. Diese Variante sollte verwendet werden wenn bei der Benutzung der Joi-Singleton-Komponente erkennbar sein soll, dass es sich um ein Singleton handelt.

Die andere Variante hat, wie eine Joi-Komponenten, ein Enum Factory und eine Klassenvariablen FACTORY vom Typ Factory. Diese Variante kann zum Beispiel hilfreich sein, wenn die Implementierung einer Komponente zu Optimierungszwecken nachträglich auf ein Singleton umgestellt werden soll.

Zur Illustration werden die Regeln, wie bei den Regeln für die Joi-Komponente, ergänzend in einem Code-Beispiel dargestellt. Die Joi-Komponente MySingletonComponent der Code-Beispiele, unterstützt

die Service Schnittstelle(Interface) MyService (bedeutet die Implementation Klasse garantiert die Implementierung der Schnittstelle MyService) und im Falle der Variante mit Factory implementiert die Factory der Joi-Singleton-Komponente MysingeltonComponent die Schnittstelle MyProvider:

2.3.1 Basisregeln für Joi-Singleton-Komponenten

- Eine Joi-Singleton-Komponente muss eine Outer-Most-Type-Klasse sein.
- Eine Joi-Singleton-Komponente muss final sein.
- Eine Joi-Singleton-Komponente darf von keiner Klasse erben.
- Eine Joi-Singleton-Komponente darf keine Schnittstelle implementieren.
- In einer Joi-Singleton-Komponente muss eine geschachtelte Enum mit dem Namen Implementation definiert werden.
- In einer Joi-Singleton-Komponente kann, als Member der Joi-Singleton-Komponente, ein Enum mit dem Namen Factory definiert werden.
- Wenn in einer Joi-Singleton-Komponente eine Enum Factory definiert wurde, dann muss die Joi-Singleton-Komponente eine Variable mit dem Namen FACTORY, als Member der Joi-Singleton-Komponente haben. Hat die die Joi-Singleton-Komponente kein Enum Factory muss die Joi-Singleton-Komponente eine Variable mit dem Namen SINGLETON haben.
- Der Default-Konstruktor einer Joi-Singleton-Komponente muss als private deklariert sein.
    - Ein guter Stil wäre es außerdem, wenn der Konstruktor als einzige Anweisung einen AssertionError() wirft (Wird nicht kontrolliert, sondern empfohlen).
- Eine Joi-Singleton-Komponente darf keine weiteren Mitglieder haben.

Folgendes Code-Beispiel stellt die Joi-Singleton-Komponente MySingletonComponent ohne Factory da. Aspekte die von den „Regeln für Joi-Singleton“ beschrieben werden, sind hervorgehoben:

```
@JoiComponent
public class MySingletonComponent ~~extends implements~~{
    private MySingletonComponent(){
        throw new AssertionError();
    }

    public static Implementation SINGLETON = Implementation.INSTANCE;

    private enum Implementation implements MyService{…
}
```

Folgendes Code-Beispiel stellt die Joi-Singleton-Komponente MySingletonComponent mit Factory da. Aspekte die von den „Regeln für Joi-Singleton“ beschrieben werden, sind hervorgehoben:

```
@JoiComponent
public class MySingletonComponent ~~extends implements~~{
    private MySingletonComponent(){
        throw new AssertionError();
    }

    private enum Factory implements MyProvider{…

    public static Factory FACTORY = Factory.INSTANCE;

    private enum Implementation implements MyService{...
}
```

2.3.1.1 Regeln für das Member Enum Implementation einer Joi-Singelton-Komponente

- Die Implementation muss private sein
- Die Implementation muss genau eine Konstante mit dem Name INSTANCE enthalten.
- Jede public Methode der Implementation muss mit @Override annotiert sein.

Folgendes Code-Beispiel stellt das Member-Enum Implementation der Joi-Singleton-Komponente MySingletonComponent da. Aspekte die von den „Regeln für das Member Enum Implementation einer Joi-Singleton-Komponente“ beschrieben werden, sind hervorgehoben:

```
…//outer class MySingletonComponent
private enum Implementation implements MyService{
    INSTANCE;

    @Override
    public void doSomething() {…
}
…//outer class MySingletonComponent
```

2.3.1.2 Regeln für das Member Enum Factory einer Joi-Singleton-Komponente

- Die Factory muss genau eine Konstante mit dem Name INSTANCE enthalten.
- Jede public Methode der Factory muss mit @Override annotiert sein.
- Jede public Methode der Factory muss den Rückgabetyp Implementation haben.
- Jede public Methode der Factory darf keine Parameter haben.
- Die Return-Anweisung jeder public Methode der Factory muss die Gestalt:
  return Implementation.INSTANCE
  haben.

Folgendes Code-Beispiel stellt das Member-Enum Factory der Joi-Singleton-Komponente MySingletonComponent mit Factory da. Aspekte die von den „Regeln für das Member Enum Factory einer Joi-Singleton-Komponente“ beschrieben werden, sind hervorgehoben:

```
…//outer class MySingletonComponent
private enum Factory implements MyProvider{
    INSTANCE;

    @Override
    public Implementation getService(<paramters>) {
        return Implementation.INSTANCE;
    }
}
…//outer class MySingletonComponent
```

2.3.1.3 Regeln für die Member-Variable FACTORY einer Joi-Komponente

- Die FACTORY muss public sein.
- Die FACTORY muss static sein.
- Die FACTORY muss final sein.
- Die FACTORY muss vom Typ Factory sein
- Die FACTORY muss mit der Factory Konstanten Factory.INSTANCE initialisiert

Folgendes Code-Beispiel stellt die Member-Variable FACTORY der Joi-Singleton-Komponente MySingletonComponent mit Factory da. Aspekte die von den „Regeln für das Member-Variable FACTORY einer Joi-Singleton-Komponente“ beschrieben werden, sind hervorgehoben:

```
…//outer class MySingletonComponent
public static final Factory FACTORY = Factory.INSTANCE;
…//outer class MySingletonComponent
```

2.3.1.4 Regeln für die Member-Variable SINGELTON einer Joi-Komponente

- Die SINGLETON muss public sein.
- Die SINGLETON muss static sein.
- Die SINGLETON muss final sein.
- Die SINGETON muss vom Typ Implementation sein.
- Die SINGLETON muss mit der Factory Konstanten Implementation.INSTANCE initialisiert werden.

Folgendes Code-Beispiel stellt die Member-Variable SINGLETON der Joi-Singleton-Komponente MySingletonComponent ohne Factory da. Aspekte die von den „Regeln für das Member-Variable SINGLETON einer Joi-Singleton-Komponente“ beschrieben werden, sind hervorgehoben:

```
…//outer class MyComponent
public static final Implementation SINGLETON = Implementation.INSTANCE;
…//outer class MyComponent
```

3 JoiLight

Beim Programmieren von Joi-Komponenten entstehen viele Code-Passagen, die sich von Joi-Komponente zu Joi-Komponente nur geringfügig unterscheiden. Solche Passagen werden auch als Boilerplate-Code bezeichnet.

Joi wurde ursprünglich als Spracherweiterung für Java mit eigenem Compiler, der Java Code generiert, entwickelt. Dem, was sich im Joi-Stil als Boilerplate code zeigt, der vom Programmierer geschrieben werden muss, entspricht in Joi zum Teil Code, der vom Joi Compiler generiert wird.

Für Programmierer ist das Schreiben von Boilerplate-Code lästiger Aufwand, der zu unreflektiertem und fehleranfälligem ‚Copy&Paste' verführt.

Einige Regeln des Joi-Stils schränken den Programmierer ein, ohne dass die Einschränkung durch einen deutlich erkennbaren Beitrag zum Erreichen des Zieles „Die Reduktion von Code-Abhängigkeiten" leisten. So eine Regel ist z.B. die Regel „Die Return-Anweisung jeder public Methode des Enums Factory muss die Gestalt: *return new Implementation(<parameter>);* haben."
Die Regel „Jede public Methode des Enums Factory muss den Rückgabetyp Implementation haben" ist völlig ausreichen, um sicherzustellen, dass Objekte der Implementation einer Joi-Komponenten zurückgegeben werden.

Um das Programmieren von Joi-Komponenten für Programmierer einfacher und übersichtlicher angenehmer zu gestalten und die Akzepttanz für den Stil zu erhöhen, wurde im Rahmen dieser Bachelorarbeit eine schlankere Version des Joi-Stils entwickelt, der auf wesentlichste Regeln reduziert wurde und den Programmierern einen möglichst hohen Freiheitgrad bei der Implementierung lässt und trotzdem dieselben Ziele verfolgt wie der Joi-Stil. Dieser schlankere Programmierstil wird als JoiLight-Stil bezeichnet.

Der JoiLight-Stil beschreibt, ebenfalls wie der Joi-Stil, das Programmieren von Komponenten. Klassen die mit der Annotation @JoiLightComponent annotiert sind, werden als JoiLight-Komponenten bezeichnet.

Statt einer Factory als Member-Enum haben JoiLight Komponenten, implizit statische, Factory-Methode (vgl. Bloch [1] S. 5ff) - Implizit statisch, da bei nicht instanziierbaren Klassen nur statische Methoden verwendet werden können.

Der JoiLight-Stil sieht keine speziellen Regeln für die Komposition oder für Singleton vor. Die JoiLight-Regel sind hinreichend offen, so dass Komposition und Singleton-Muster mit JoiLight-Komponenten realisiert werden können.

Zur Illustration werden die Regeln ergänzend in einem Code-Beispiel dargestellt. Die JoiLight-Komponente MyLightComponent der Code-Beispiele, unterstützt die Service Schnittstelle(Interface) MyService (bedeutet die Implementation Klasse garantiert die Implementierung der Schnittstelle MyService) aus den vorhergehenden Code Beispielen.

3.1 Regeln für JoiLight-Komponenten.

- Eine JoiLight-Komponente muss final sein.
- Eine JoiLight-Komponente darf von keiner Klasse erben.
- Eine JoiLight-Komponente darf keine Schnittstelle implementieren.
- Der Default-Konstruktor einer JoiLight -Komponente muss als private deklariert sein.
    - Ein guter Stil wäre es außerdem, wenn der Konstruktor als einzige Anweisung einen AssertionError() wirft. Dies wird nicht kontrolliert, sondern empfohlen.
- Jede Variable(Attribute und Klassenvariablen) muss private sein.
- Jede nicht private Methode der JoiLight -Komponente muss den Rückgabetyp Implementation haben.
- In einer JoiLight -Komponente muss eine geschachtelte Klasse mit dem Namen Implementation definiert werden.
- Eine Joi-Komponente darf keine weiteren Mitglieder haben.

Hinweis: Es können nur statische Methoden und Variablen einer MyLightComponente innerhalb und außerhalb einer JoiLight-Komponente aufgerufen werden, da diese nicht instanziierbar ist.

Es werden die Namenskonventionen für statische Factory Methoden von Joshua Bloch (siehe Bloch [1] S. 10) empfohlen.

Folgendes Code-Beispiel stellt die JoiLight-Komponente MyLightComponent da. Aspekte die von den „Regeln für JoiLight-Komponenten.“ beschrieben werden, sind hervorgehoben:

```
@JoiComponent
final class MyLightComponent extends implements{

    private MyLightComponent() {
        throw new AssertionError();
    }

    public static Implementation getInstance(){
        return new Implementation();
    }

    private static final class Implementation implements MyService {…
}
```

3.1.1 Regeln für die Member-Klasse Implementation einer JoiLight Komponente

- Die Klasse Implementation muss private sein.
- Die Implementation muss static sein.
- Die Implementation muss final sein.
- Die Implementation darf von keiner Klasse erben.
- Jede public Methode der Implementation muss mit @Override annotiert sein.

Folgendes Code-Beispiel stellt die Member-Klasse Implementation JoiLight-Komponente MyLightComponent da. Aspekte die von den „Regeln für JoiLight-Komponenten.“ beschrieben werden, sind hervorgehoben:

```
…//outer class MyLightComponent
private static final class Implementation ~~extends~~ implements MyService {
    …
    @Override
    public void doSomething() {…
     ... // other service-specific members go here
}
…//outer class MyLightComponent
```

4 Elemente Kennzeichnen mit Annotationen und Namenskonventionen

Bei der Beschreibung von Codestrukturen, wie z.B. Joi-Komponenten, ist es erforderlich Elemente oder Codeabschnitte, die für die beschriebene Struktur eine syntaktische Bedeutung haben, zu kennzeichnen, damit Tools die Struktur analysieren und z.B. auf Korrektheit prüfen können. Auch das Lesen und Verstehen des Quellcodes ist für Programmierer leichter, wenn die Elemente die zu einer Struktur gehören, klar in ihrer Funktion erkennbar sind. Programmiersprachen (oder auch Spracherweiterungen wie Joi) verwenden in der Regel Schlüsselwörter für diese Kennzeichnung.

Für einen Programmierstil, steht das Mittel der Definition zusätzlicher Schlüsselwörter im engeren Sinne nicht zur Verfügung. Deshalb müssen andere Mittel zur Kennzeichnung verwendet werden.

Eine Klassische Technik ist die Verwendung von Namenskonventionen, also der Benennung von normalen Sprachelementen wie Typ- oder Methoden-Definitionen nach bestimmen Mustern

Seit der Java Version 5.0 ist es möglich, in Java Sprachelemente durch Annotationen zu kennzeichnen, bzw. Metadaten zu den Elemente bereitzustellen. Annotationen eigenen sich sehr gut um zusätzliche Informationen, wie z.B. die Syntaktische Bedeutung eines Java-Elementes in einem Programmierstils abzulegen. Die Verwendung von Annotationen macht die klassische Technik spezieller Kommentare für die maschinelle Auswertung obsolet.

Die Verwendung von Annotationen zu Kennzeichnung von Elementen hat gegenüber der Verwendung Namenskonventionen einige Vor und Nachteile, die im nachfolgendem Abschnitt kurz erläutert werden.

Der Joi-Stil setzt beide Techniken ein, um die Joi-spezifischen Stil-Elemente (Joi-Komponente, Factory, FACTORY, Implementation…) zu kennzeichnen.

Bloch (vgl. Bloch[1] S. 196ff) empfiehlt Annotationen anstelle von Namenskonventionen zur Kennzeichnung von Strukturelemente, die von Tools oder Framework wie Checkstyle ausgewertet werden, zu nutzen, da diese viele Vorteile gegenüber Namenskonvention ,insbesondere solchen die eine Namenmuster beschreiben, haben.

Einiger der Vorteile von Annotationen gegenüber Namenskonventionen sind:

- Die Korrekte Schreibweise von Annotationen wird im Gegensatz zu Namenskonventionen vom Compiler überprüft.
- Annotationen können in der Deklaration auf bestimmte Elementtypen eingeschränkt werden, so dass der korrekte Einsatz vom Compiler überprüft wird.
- Annotationen können durch Variable wesentlich mehr Metainformationen zu einem Element bereitstellen, als dieses mit Namenskonventionen sinnvoll möglich ist. Auch können einzelne Informationen gezielt ausgelesen werden.
- Soll ein Element mehrfach gekennzeichnet werden, tendieren Namenskonventionen zu langen und umständlichen Namen. Dasselbe gilt, wenn es mehrere Elemente mit derselben Kennzeichnung, im selben Namensraum geben darf, da dann der Name aus einem fixen und einem variablen Teil besteht müssen.
- Elemente können auch versehentlich konform zu einer Namenskonvention benannt werden.
- Annotationen können zur Laufzeit per Reflektion ausgelesen werden.

Aber auch Namenskonventionen haben gegenüber Annotationen Vorteile:

- Elemente, die einer Sprachkonvention unterliegen, erleichtern Programmierern das Lesen und Verstehen von Quellcode, sowie das verwenden der Elemente, da die Elemente überall durch ihren Namen gekennzeichnet sind.
- Annotierte Elemente sind nur genau an der Stelle im Dokument gekennzeichnet wo es annotiert werden, was meisten bei der Definition der Elemente stattfindet. Programmierer die wissen wollen ob ein Element durch eine Annotation gekennzeichnet ist müssen nach dieser Stelle suchen, oder mithilfe von Reflektion diese erst zur Laufzeit auslesen.

Der Joi-Stil sieht zur Kennzeichnung der Strukturelemente sowohl Annotationen wie auch Namenskonventionen vor, um sowohl das Lesen und Programmieren von Joi-Komponenten, als auch das Überprüfen durch Checkstyle möglichst einfach zu machen.

Durch das Annotieren einer Klasse mit den Annotationen @JoiComponent oder @JoiLightComponent wird diese als Joi-Komponente, bzw. als JoiLight-Komponente gekennzeichnet, und muss den Regeln des Joi- bzw. des JoiLight-Stils entsprechen.

Eine Namenskonvention für die Joi-Komponenten, bzw. der Klasse die eine Joi-Komponente sein soll, würde geg. zu langen Klassenamen (Komponentennamen) führe sich die Namen aus einen einem festen Teil z.B. JoiComponent und einen Variablen Teil zusammensetzen müssen, um mehre Klassen im selben Package(Namensraum) definieren zu können. Ein Klassenname für eine Joi-Komponente könnte zu Beispiel JoiComponentMyService sein. Hinzukommt das bei einem Rechtschreibfehler im fixen Teil des Namen dazu führen würde, das zwar der Compiler die Klasse im Sinne der Java-Grammatik als korrekt einstuft, Checkstyle könnte aber die Klasse gar nicht erst als Joi-Komponente identifiziert, folglich er weise auch nicht gegen die Joi-Regeln prüfen. Abhilfe würde eine Regel schaffen, die es erzwing alle das (Outer-Most-)Klassen Joi-Komponenten, bzw. Teil einer Joi-Komponenten, seine müssen, was eine Kennzeichnung der Outer-Most-Klasse überflüssig machen würde(Es müssten noch weitere Regeln definiert werden, um das Implementieren von Klassen mit main Methode zu ermöglichen).

Innerhalb einer Joi-Komponente werden verschiedene Elemente durch Namenskonventionen gekennzeichnet. Dieses hat mehrere Vorteile gegenüber dem Annotieren dieser Elemente. Der Quellcode von Joi-Komponenten wird so besser lesbar. Einige Prüfungen sind mit Checkstyle etwas einfacher zu realisieren. Der Nachteil, dass Schreibfehler nicht auffallen, trifft in unserem Fall nicht, da von den Elementen für die eine Namenskonvention jeweils genau eins vorkommen darf, führt jeder Schreibfehler zur Verletzung einer Regel.

Da alle Elemente einer Joi Komponente, für die Namenskonventionen bestehen, innerhalb jeder Joi-Komponente eindeutig sein müssen, bieten sich Namenskonventionen an, die für jedes Strukturelement einen fixen Namen vorsehen. So ist es nicht möglich zwei unterschiedliche Elemente mit der gleichen Kennzeichnung zu versehen. Bei Annotationen müsste dies zusätzlich von Checkstyle geprüft werden.

5 Implementierung der Überprüfung des Joi-Stils mit Checkstyle

5.1 Checkstyle

Checkstyle ist ein statisches Codeanalyse-Tool mit dem Java Quellcode Dateien auf die Einhaltung von Programmierrichtlinien automatisiert überprüft werden können. (vgl. Burn[2])

Checkstyle kann über Kommandozeile oder als ANT-Task in einem Build-Prozess gestartet werden. Zusätzlich sind Plug-ins zut Integration von Checkstyle für diverse Entwicklungsumgebungen verfügbar,

Checkstyle wird mit Programmiersprache Java entwickelt und ist Open Source. Checkstyle wird unter der `GNU LESSER GENERAL PUBLIC LICENSE Version 2.1` vom Februar 1999 veröffentlicht.

Checkstyle kann über selbstprogrammierte FileSetCheck Module und Submodule des Treewalkers erweitert werden.

Checkstyle ist modular aufgebaut. Die Module können durch Konfiguration kombiniert werden. Einige Module bieten Schnittstellen zur Erweiterung. Die Module bilden eine Hierarchie. Auf oberster Ebene liegt das Modul Checker, das die zu prüfenden Dateien einliest. Auf der nächsten Ebene liegen Module die das Interface FileSetCheck implementieren und die eingelesenen Dateien vom Checker übergeben bekommen. Das Interface FileSetCheck definiert Methodensignaturen zum Entgegennehmen von Dateien und Werfen von Fehlern. Zu diesen gehört der Treewalker. Die meisten Checks (Prüfungen von Programmierrichtlinien) sind als Submodule des Treewalker implementiert.

Der Treewalker erzeugt aus einzelnen Java Dateien (.java), die es vom Checker übergeben bekommt, mithilfe des ANTLR Parsers, je einen AST(Abstract Syntax Tree). Dabei wird jedes Literal im Quellcode, als eigener Token im AST abgebildet. Zusätzlich gibt es noch einige „Container-Token“ die logisch zusammenhängende Token gruppieren. Jeder Token hat einen Typ der die Syntaktische Bedeutung des Tokens wiedergibt.

Über Konfigurationsdateien werden zur Ausführungszeit Submodule beim Treewalker für Token-Typen registriert. Die Submodule des Treewalkers (Checks) sind Klassen, die von der abstrakten Klasse Check u.a. die Methoden *public void visitToken(DetailAST aAST)* und *public abstract int[] getDefaultTokens()* erben und überschreiben.

Der Treewalker traversiert über den AST und ruft dabei an den Token für die für den Token-Typ des Token registrierten Checks. die Methode *visitToken(aAST)* auf. Die Checks überprüfen den AST, meist in der näheren Umgebung des übergebenen Tokens, gegen die Regeln die sie implementieren.

Die Klasse DetailAST, die den AST implementiert, stellt unter anderem Methoden zum traversieren des Ast bereit, die auch in den Checks verwendet werden können.

5.2 Konzept für Joi-Stil Prüfungen

Bei typischen Prüfungen, die von Checkstyle Modulen geprüft werden, genügt es, den AST in der unmittelbaren Umgebung des Tokens, für das eine Prüfmethode aufgerufen wurde zu analysieren.

Für einige Joi-Prüfungen genügt das nicht, da der Joi-Stil Regeln enthält, die Konsistenzforderungen zwischen Objekten auf unterschiedlichen Ebenen der Member-Hierarchie stellen. Dies gilt insbesondere für die Prüfung der Joi-Komposition. Bei der Implementierung solcher Konsistenzprüfungen sind auch die Möglichkeiten der Schachtelung von Klassen zu beachten.

Ein Aufbau von Konsistenzprüfungen zwischen Objekten über Ebenen der Member-Hierarchie aus lokalen Prüfungen des AST, die vom Treewalker unabhängig voneinander für geeignete Token-Typen aufgerufen werden, ist schwierig, da die relevante Beziehung gefundener Token zueinander wird nur über einen Weg im AST ersichtlich wird.

Das im Folgenden beschriebene Konzept ermöglicht die für Joi erforderlichen Prüfungen. Es kann auf die Prüfung anderer Programmierstile oder Pattern, die Member-Hierarchie übergreifende Prüfungen erfordern, übertragen werden.

5.2.1 Checkstyle-Integration

Alle Prüfungen auf Einhaltung Regeln des Joi-Stils (des JoiLight-Stils) sind in einer Klasse realisiert. Hintergrund ist zum einen das im folgenden Abschnitt dargestellte Konzept, das die Konsistenzprüfungen über die Member-Hierarchie hinweg erlaubt. Zum anderen wird dadurch die Konfiguration der Prüfungen in Checkstyle erleichtert. Da so der ganze Satz von Prüfungen einfach in einem Schritt in Checkstyle konfiguriert werden kann.

Die Prüfungen für die Regeln des Joi-Stils sind in der Java Klasse JoiCheck realisiert. Die Prüfungen für die Regeln des JoiLight-Stils sind in der Klasse JoiLightCheck realisiert.

Die Klassen JoiCheck und JoiLightCheck erben von der abstrakten (Checkstyle-)Klasse Check und überschreiben jeweils die Methoden visitToken(DetailAST aAST) und getDefaultToken(). So können sie als Submodul des Treewalkers in Checkstyle verwendet werden.

Die Klassen JoiCheck bzw. JoiLightCheck werden für CLASS_DEF-Token registriert.

Die gewählte Checkstyle Integration ist nur zur Überprüfung einzelner Dateien geeignet. Da alle Joi Regeln sich auf Joi Komponenten beziehen und der Quell-Ciode für eine Joi Komponenten in einer Datei abgelegt wird, ist das für die Prüfung des Joi-Stil keine Einschränkung.

5.2.2 Schalten der Checks

Durch das Kennzeichnen einer Klasse mit den Annotationen @JoiComponent oder @JoiLightComponent wird angezeigt, dass die Implementierung den Regeln des Joi-Stils bzw. des Joi-Light-Stils folgen soll.

Die Existenz der Annotation wird in der Realisierung der Checkstyle-Erweiterung als Schalter für die Aktivierung der Regeln verwendet. Wenn die Methode *visitToken(DetailAST aAST)* vom Treewalker für ein CLASS_DEF-Token aufgerufen wird, wird im AST überprüft, ob die dem Token entsprechende Klassendefinition mit @JoiComponent (bzw. JoiLightComponent) annotiert ist. Wenn dies der Fall ist,

kann vorausgesetzt werden, dass das CLASS_DEF -Token zu einer Joi-Component (bzw. JoiLight-Component) Klasse gehört und die Klasse wird im AST gegen die Joi-Regeln (bzw. die Joi-Light-Regeln) validiert.

Die Umschaltung der Prüfungen zwischen den Varianten des Joi-Stils erfolgt implizit nach folgenden Regeln:

- Enthält die Joi-Component-Klasse eine Member-Klasse Forwarding, werden die Regeln für die Prüfung der Joi-Komposition aktiviert
- Enthält die Joi-Component-Klasse ein Enum Implemenation werden die Regeln für die Prüfung der Joi-Singleton-Variante aktiviert
- Enthält die Joi-Component-Klasse ein Enum Factory und ein Enum Implementation so werden die Regeln für die Joi-Singleton-Variante mit Factory aktiviert, sonst gelten die Regeln der Joi-Singleton-Variante ohne Factory.

5.2.3 Phasenkonzept für Prüfungen

Die Prüfungen für alle Teile einer Joi-Komponente – mit @JoiComponent-annotierte Klasse, Factory, Implementation-Klasse oder Implementation-Enum und gegebenenfalls Forwarding-Klasse etc. – laufen innerhalb des Aufrufs der Methode visitToken für das CLASS_DEF-Token der Joi-Component-Klasse ab.

Die Prüfungen gegen die Regeln des Joi-Stils oder genauer gegen die Regeln der Varianten des Joi Stils erfolgen geschachtelt nach Member-Hierarchie in jeweils drei Phasen:

5.2.3.1 Phase 1

In der ersten Phase werden die Basis Regeln für die Joi-Komponente geprüft, die lokal durch Analyse des Deklarationsteils erfolgen können.

Die Joi-Stil-Regel „Eine Joi-Komponente muss final sein“, ist ein Beispiel für eine Regel, die in Phase 1 geprüft wird.

5.2.3.2 Phase 2

In der zweiten Phase werden die Member-Objekte der Joi-Component Class einzeln geprüft.

Die Member Objekte werden auf ihren Java-Schlüsselwort-Typ hin untersucht. Falls für den Typ Regeln definiert sind, wird das Objekt in eine Prüfmethode für diesen Typ entsprechenden Prüfung unterzogen.

Ein Beispiel für eine Regel, die in der Phase 2 bei der Klasse Forwarding geprüft wird, ist die Regel „Alle public Methoden der Klasse Forwarding müssen mit @Override annotiert sein“.

Wenn die Regeln dies erfordern, erfolgt die Prüfung eines Member- Objekts in Phase 2 wieder analog zur Gesamtprüfung in 3 Phasen (‚Schachtelung nach Member-Hierarchie) wobei nun in der zweiten Phase die Member-Objekte des Member-Objekts –also sozusagen die Enkel-Objekte des ursprünglichen Objekts geprüft werden.

Ein Beispiel für eine Regel, bei der die die Schachtelung greift, ist die Regel über die Gestalt der Methoden der Klasse Forwarding für die Joi-Komposition.

5.2.3.3 Phase 3

In der dritten Phase wird die Konsistenz der Ergebnisse untereinander geprüft.

Wenn es das Member Enum Factory gibt, muss es als Member eine öffentliche Klassenvariable FACTORY vom Typ Factory geben geben. Wenn es kein Member Enum Factory gibt, muss es eine öffentliche Klassenvariable SINGLETON vom Typ Implementation geben.

Weitere Beispiele für Phase 3 Prüfungen sind die Regeln der Joi-Komposition, bezüglich der Konsistenz zwischen den Methoden der Forwarding Klasse und den Signaturen in der Schnittstelle Forward.

Die Gliederung in die drei Phasen erleichtert die Pflege und Erweiterung des Prüfmoduls, da die Analyse und Prüfung der einzelnen Objekte von der übergreifenden Konsistenzprüfung getrennt wird. Die Einzelprüfungen haben so untereinander keine Abhängigkeiten, insbesondere ist die Reihenfolge der Einzelprüfungen unerheblich.

Für die Kombination der Prüfungen in der zweiten und dritten Phase kann ein vereinfachtes Analogon zum Blackboard Pattern verwendet werden: In der zweiten Phase prüfen Spezialisten die einzelnen Member-Objekte und stellen ihr Ergebnis in einem (einfachen) Blackboard zur Verfügung. In der dritten Phase werden die Member-Objekt-übergreifenden Konsistenzprüfungen aufgrund der Eintragungen im Blackboard ausgeführt. (vgl. Hoschule für Technik Rapperswil[7])

Die üblichen Problem-Charakteristika für einen Einsatz des Blackboard-Pattern wie ‚vollständige deterministische algorithmische Lösung nicht möglich oder zu aufwändig' sind in unserem Fall offensichtlich nicht erfüllt. Das Blackboard-Pattern bietet aber eine starke Entkopplung einzelner algorithmischer Teillösungen voneinander. Dieser Aspekt ist auf unsere Situation übertragen interessant. Es ist von Vorteil, wenn Routinen, die Äste des AST auf Teilaspekte von Regeln hin untersuchen, möglichst stark entkoppelt voneinander arbeiten. Insbesondere wird so eine Anpassung der Prüfmodule an Änderungen des Regelwerks erleichtert.

6 Fazit

Das Ergebnis der Arbeit kann abschließend kurz zusammengefasst werden:

- Die Unterstützung von JOI als Programmierstil mit Checkstyle ist möglich.
- Bei der Umsetzung wurden Annotationen gegenüber Namenskonventionen weniger intensiv eingesetzt als ursprünglich erwartet.
- Zur Verbesserung der Praxistauglichkeit ist eine vereinfachte Form von Joi als Programmierstil wie der JoiLight-Stil sinnvoll.

Die Arbeit hat gezeigt, dass Joi in geeigneter Weise als Programmierstil (Joi-Stil) formuliert werden kann, so dass die Einhaltung der Regeln des Joi-Stils mithilfe einer Erweiterung des Tools Checkstyle automatisiert überprüft werden kann.

Das gewählte Verfahren für die Implementierung des Prüf-Tools ist auf die automatisierte Prüfung anderer Java-Programmierstile und geeigneter Design-Pattern übertragbar.

Checkstyle hat sich durch seine Offenheit und Erweiterbarkeit als geeigneter Rahmen für die Implementierung des Prüftools für den Joi-Stil erwiesen. Für die Überprüfung des Joi-Stils wird nur ein verhältnismäßig kleiner Teil der Funktionalität von Checkstyle genutzt. Der Weg, die Joi-Stil Prüfungen als Submodule des Treewalkers zu implementieren, wurde gewählt, obwohl die Kern-Funktionalität des Treewalkers, das Traversieren über den AST, kaum genutzt wird, da dieser Weg für den Nutzer eine komfortable Einbindung in Checkstyle in gewohnter Form bietet.

Eine Voraussetzung für die automatische Prüfung ist, dass relevante Strukturelemente durch Annotationen oder Namenskonventionen gekennzeichnet werden. Für den Joi-Stil hat es sich als praktikabel erwiesen, für die Kennzeichnung der Joi-Komponenten eine Annotation zu verwenden, die zum Aktivieren der Prüfung in Checkstyle genutzt wird. Die Strukturelemente innerhalb der JOI Komponenten werden über Namenskonventionen gekennzeichnet. Dies ist insbesondere dadurch praktikabel, dass beim Joi Stil die Namen nicht nur einer Bildungsregel folgen müssen, sondern vielmehr fest vorgegeben werden können. So ergibt sich zum einen eine gute Lesbarkeit des Quell-Codes zum anderen wird die automatische Prüfung erleichtert.

Beim Formulieren des Joi-Stils hat sich gezeigt, dass die beschriebenen Komponentenstrukturen einen größeren Anteil Boilerplate-Code enthalten. Joi wurde ursprünglich als Spracherweiterung für Java mit eigenem Compiler, der Java Code generiert, entwickelt. Dem, was sich im Joi-Stil als Boilerplate code zeigt, der vom Programmierer geschrieben werden muss, entspricht in Joi zum Teil Code, der vom Joi Compiler generiert wird. Für Programmierer ist das Schreiben von Boilerplate-Code lästiger Aufwand, der zu unreflektiertem und fehleranfälligem ‚Copy&Paste' verführt.

Einige der Regeln des Joi-Stils schränken den Programmierer ein, ohne einen wesentlichen Vorteil im Hinblick auf die angestrebte Reduzierung von Code-Abhängigkeiten zu bewirken.

Die Ergänzung des Joi-Stils für die Joi-Komposition als Ersatz für Vererbung mit Regeln für Forward-Schnittstelle und Forwarding-Klasse wirkt insgesamt recht kompliziert. Insofern ist fraglich, ob diese Ergänzung hinreichende Akzeptanz finden wird. Für den Aspekt, die Komposition gegen die Vererbung zu propagieren, erscheinen die Randbedingungen für die Variante Joi als Sprachstil im Vergleich zu den Bedingungen für Joi als Spracherweiterung mit Compiler ungünstiger.

Aufgrund der Erfahrungen mit der Formulierung der Regeln des Joi Stils in einer für die Prüfung geeigneten Form und der Implementierung der Prüfmodule wurde der JoiLight-Stil als Vereinfachung des Joi-Stils entwickelt. Der Joi Stil hat mit seinen wenigen einfachen Regeln, die ein unübersichtliches Aufblähen des Codes vermeiden, im Vergleich zum Joi-Stil sicher eine bessere Grundlage für Akzeptanz bei Programmierern als der Joi-Stil. Hinsichtlich der Ziels Codeabhängigkeiten zu vermeiden, erreicht der JoiLight-Stil fast die gleiche Wirkung wie der Joi-Stil.

7 Literaturverzeichnis

1. **Bloch Joshua:** Effective Java: Programming Language Guide. Second Edition., Addison-Wesley; 2008

2. **Burn Oliver:** Checkstyle 5.1. Checkstyle http://checkstyle.sourceforge.net/; letzer Zugriff 11. Juni 2010

3. **Drachenfels Heiko von:** Komponentenorientierte Programmierung im Kleinen“; Informatik-Spektrum 28(2); S. 136-143; Springer; April 2005

4. **Drachenfels Heiko von, Haase Oliver, Walter Robert:** Joi - Eine Java-Spracherweiterung zur Reduzierung von Codeabhängigkeiten; HTWG FORUM; S. 54-57; 2008/2009

5. **Gamma Erich, Helm Richard, Johnson Ralph B., Vlissides John**: Design Patterns. Elements of Reusable Object-Oriented Software., Addison Wesley 1995

6. **Gosling James, Joy Bill, Steel Guy, Bracha Gilad**: The Java Language Specification. Third Edition; Addison-Wesley; 2005

7. **Hoschule für Technik Rapperswil, Institut für Software**: Blackboard, http://wiki.hsr.ch/APF/wiki.cgi?Blackboard; Letze Änderung 4. November 2009; Letzter Zugriff 17. Juni 2010

8. **Vashishtha ShriKant; Gupta Abhishek:** Automated code reviews with Checkstyle, Part 1. JavaWorld.com. http://www.javaworld.com/javaworld/jw-11-2008/jw-11-checkstyle1.html?page=1. Letze Änderung 25. November 2008; Letzer Zugriff 11. Juni 2010.]

Anhang A Quellcode Dateien mit Checkstyle und den JoiCheck und JoiLight Check Modul überprüfen

In diesem Abschnitt wird erklärt, wie man Java Quell-Code Dateien mit Checkstyle und den im Rahmen dieser Bachelorarbeit entwickelten Modulen JoiCheck und JoiLightCheck, sowie den Annotationen @JoiComponent und @JoiLightComponent überprüfen kann.

Installation

Der einfachste Weg, Checkstyle zu nutzen ist, die Jar-Datei checkstyle-all-5.1.jar in den Java Classpath zu integrieren. Die Jar Datei enthält, alle notwendigen Dateien um Checkstyle mit den Standardmodulen zu nutzen. Die Jar Datei checkstyle-5.1.jar, kann von der Checkstyle Homepage http://checkstyle.sourceforge.net/ herunterladen.

Um die Module JoiCheck und JoiLight Check nutzen zu können, muss außerdem die Jar Datei joi4checkstyle-all.jar in Classpath eingefügt werden.

Die Datei joi4checkstyle-all befindet sich auf der CD im Anhang dieser Bachelorarbeit im Ordner joi4checkstyle/ und enthält neben den beiden Modulen JoiCheck und JoiLightCheck die Annotationen @JoiComponent und @JoiLightComponent.

Für die Entwicklung mit Eclipse (oder einer anderen IDE) kann man die Jar. Datei joi4checkstyle-annotations.jar als Library(lib/), welche nur die beiden Annotationen enthält, zum Buildpath hinzufügen, um die Joi Annotationen nutzen (importieren) zu können.

Importieren der Annotationen @JoiComponent und @JoiLightComponent.

Um Klassen mit den Annotationen @JoiComponent bzw. @JoiLightComponent, als Joi-Komponenten bzw. JoiLight-Komponente markieren zu können, muss man diese mit dem Befehl: import de.htwg_konstanz.joi.annotations.JoiComponent; bzw. import de.htwg_konstanz.joi.annotations.JoiLightComponent; importieren.

Konfigurieren

Um mit Checkstyle Quell-Code Dateien überprüfen zu können, müssen alle Module(Checks), die zur Prüfung herangezogen werden sollen, in einer XML Datei konfiguriert werden.

Eine Ausführliche Anleitung zur Konfiguration findet man auf der Checkstyle Homepage unter folgendem Link http://checkstyle.sourceforge.net/config.html .

Um die Module JoiCheck und JoiLightCheck in eine Konfiguration einzubinden, muss man das Tag <module name="JoiCheck"/>, bzw. <module name="JoiLightCheck"/>, als Kindelement (Submodul) des Treewalkers in der XML Konfigurationsdatei einfügen.

Folgende Beispielkonfiguration demonstriert, wie man die Module JoiCheck und JoiLightCheck in eine Konfiguration integriert:

```
<module name="Checker">
    <module name="TreeWalker">
        <module name="JoiCheck"/>
……< module name="JoiLightCheck"/>
    </module>
</module>
```

Falls man mit einer älteren Java Version vor Java 6 arbeiet, sollte die Property javaSixIncompatible auf true gesetzt werden.

```
<module name="Checker">
  <module name="TreeWalker">
    <module name="JoiCheck">
      <property name=" javaSixIncompatible" value="true"/>
   </module>
   <module name="JoiLightCheck">
      <property name=" javaSixIncompatible" value="true"/>
    </module>
  </module>
</module>
```

Checkstyle über Kommandozeile starten

Checkstyle kann über Kommandozeilen-Befehle gestartet werden.

Um Checkstyle mit den Submodulen JoiCheck und JoiCheckLight zu starten und Java Dateien zu überprüfen, kann man folgenden Befehl nutzen:

java com.puppycrawl.tools.Checkstyle.Main \ -o <outputFile> \ -c <configurationFile> -r <dirOrFile2Check>

-o outputFile – spezifiziert die Datei in die das Ergebnis (Logmeldungen) geschrieben werden soll.

-c configurationfile – gibt an, welche XML Konfigurationsdatei verwendet werden soll.

-r dirOrFile2Check gibt an, welche java Quellcode Dateien (.java) mit Checkstyle überprüft werden sollen. Wenn der Pfad zu einem Ordner angegeben wird, werden alle darin enthaltenen Java Dateien geprüft.

Eine ausführliche Anleitung zur Nutzung von Checkstyle über Kommandozeile findet sich auf der Homepage von Checkstyle unter http://checkstyle.sourceforge.net/cmdline.html

Checkstyle als ANT-Task nutzen

Checkstyle lässt sich zusammen mit den Modulen JoiCheck und JoiLight-Check auch als ANT-Task einbinden. Eine ausführliche Anleitung findet man auf der Homepage von Checkstyle unter http://checkstyle.sourceforge.net/anttask.html .

Anhang B CD

Auf der CD im Anhang dieser Arbeit, befinden sich folgenden Dateien

- joi4checkstyle/
  - doc/
  - joi4checkstyle-all.jar
  - joi4checkstyle-annotations.jar
  - LICENCE
  - LICENCE.apache20
  - preform_joi_checks.bat
  - README
  - RIGHTS.checkstyle
  - RIGHTS.antlr
- joi4checkstyle-src/
  - src/
    - checkstyle/
      - com/puppycrawl/tools/checkstyle/checks/joi/
        - JoiCheck.java
        - JoiLightCheck.java
        - messages.properties
        - package-info.java
    - testinput/
      - com/puppycrawl/tools/checkstyle/joi/
      - com/puppycrawl/tools.checkstyle.joilight/
    - test/
      - com/puppycrawl/tools/checkstyle/checks/joi/
        - JoiCheckTest.java
        - JoiLightCheckTest.java
    - de/htwg_konstanz/joi/annotations
        - JoiComponent.java
        - JoiLightComponent.java
    - README
- testreports/
  - emma
  - testreports
- BA_JMiddendorf.pdf

Ehrenwörtliche Erklärung

Hiermit erkläre ich, Jonas Middendorf, geboren am 02. Juli 1985 in Hannover,

(1) dass ich meine Bachelorarbeit mit dem Titel:

„Unterstützung von Joi als Programmierstil mit Annotationen und Checkstyle“
“

in der Fakultät Informatik an der HTWG-Konstanz unter Anleitung von Professor Dr. Heiko von Drachenfels und Professor Dr. Oliver Haase, selbständig und ohne fremde Hilfe angefertigt habe und keine anderen als die angeführten Hilfen benutzt habe;

(2) dass ich die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 18. Juni 2010 Jonas Middendorf